Vue.js 2

Wprowadzenie dla profesjonalistów

—

Adam Freeman

Helion

apress

Tytuł oryginału: Pro Vue.js 2

Tłumaczenie: Krzysztof Rychlicki-Kicior

ISBN: 978-83-283-5498-2

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/vue2wp.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/vue2wp
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

ROZDZIAŁ 9.

Jak działa Vue.js?

Na początku swojej przygody z Vue.js możesz poczuć się przytłoczony dużą liczbą zagadnień do zrozumienia. Nie wszystko wydaje się mieć sens od samego początku. W tym rozdziale objaśniam, jak działa Vue.js, i pokazuję, że nie ma w nim żadnej magii — warto o tym pamiętać, gdy będę zagłębiał się w tajniki poszczególnych zagadnień Vue.js w kolejnych rozdziałach. Tabela 9.1 podsumowuje bieżący rozdział.

Tabela 9.1. *Podsumowanie rozdziału*

Problem	Rozwiązanie	Listing
Utwórz prostą aplikację webową bez pomocy Vue.js.	Skorzystaj z API **DOM** (ang. *Document Object Model* — obiektowy model dokumentu).	9.6
Utwórz prostą aplikację webową z pomocą Vue.js.	Utwórz obiekt `Vue` i skonfiguruj go przy użyciu właściwości `el` i `template`.	9.7
Zaprezentuj wartość danych użytkownikowi.	Zdefiniuj właściwość danych i użyj wiązania danych.	9.8
Zareaguj na działania użytkownika za pomocą zdarzeń.	Skorzystaj z dyrektywy `v-on`.	9.9
Wygeneruj wartości danych, które wymagają obliczeń.	Zdefiniuj właściwość obliczaną.	9.10
Utwórz wielokrotnego użytku funkcjonalną jednostkę aplikacji.	Utwórz i zastosuj komponent.	9.11 – 9.13
Zdefiniuj szablon komponentu jako kod HTML.	Skorzystaj z elementu `template`.	9.14 – 9.16

Przygotowania do rozdziału

Aby utworzyć przykładowy projekt, otwórz wiersz poleceń, przejdź do wybranego katalogu i wykonaj polecenie z listingu 9.1. To polecenie wymaga posiadania narzędzi z rozdziału 1. Musisz wykonać opisane w nim operacje, aby utworzyć projekt.

Listing 9.1. *Tworzenie przykładowego projektu*

```
vue create nomagic
```

■ **Wskazówka** Przykładowy projekt do tego rozdziału — podobnie jak do wszystkich innych — można pobrać z serwera FTP wydawnictwa Helion pod adresem *ftp://ftp.helion.pl/przyklady/vue2wp.zip*.

Gdy zostaniesz poproszony o wybór ustawień, wybierz opcję `default`. Projekt zostanie utworzony, a pakiety do wszystkich narzędzi deweloperskich zostaną pobrane i zainstalowane, co może trochę potrwać.

■ **Uwaga** W trakcie pisania niniejszej książki pakiet `@vue/cli` był dostępny w wersji beta. W związku z możliwymi zmianami warto zapoznać się z erratą dostępną w repozytorium oryginalnego wydania książki pod adresem *https://github.com/Apress/pro-vue-js-2*.

Po utworzeniu projektu do katalogu *nomagic* dodaj plik *vue.config.js* o zawartości jak w listingu 9.2. Plik ten jest używany do konfigurowania narzędzi deweloperskich Vue.js, które opisuję w rozdziale 10.

Listing 9.2. *Zawartość pliku nomagic/vue.config.js*

```
module.exports = {
    runtimeCompiler: true
}
```

Dodawanie frameworka Bootstrap CSS

Aby dodać pakiet Bootstrap CSS do projektu, wykonaj polecenie z listingu 9.3 w katalogu *nomagic*. Jest to framework CSS używany do stylowania treści HTML w tym rozdziale.

Listing 9.3. *Dodawanie pakietu Bootstrap*

```
npm install bootstrap@4.0.0
```

Po zakończeniu instalacji otwórz plik *src/main.js* i dodaj instrukcje z listingu 9.4.

Listing 9.4. *Dodawanie Bootstrapa w pliku src/main.js*

```
import Vue from 'vue'
import App from './App.vue'
import "bootstrap/dist/css/bootstrap.min.css";
Vue.config.productionTip = false
new Vue({
  render: h => h(App)
}).$mount('#app')
```

Ta instrukcja dołączy style CSS Bootstrapa do treści przesyłanych do przeglądarki.

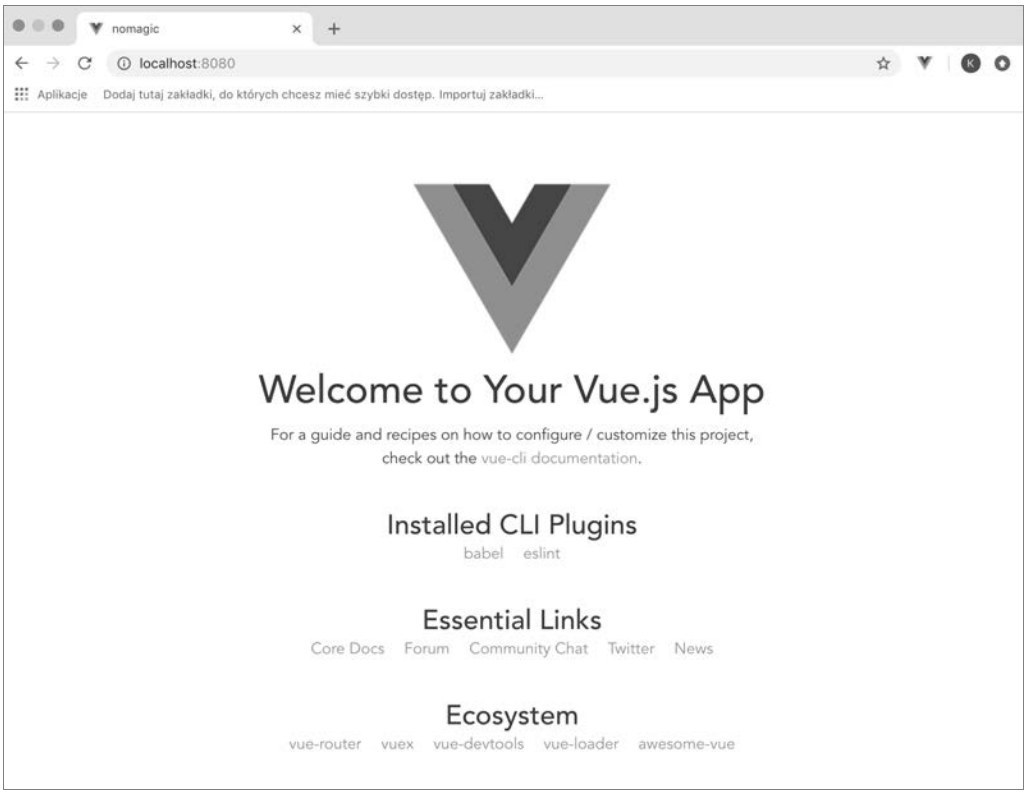
Uruchamianie przykładowej aplikacji

Projekt utworzony za pomocą polecenia z listingu 9.1 dołącza narzędzia wymagane w procesie tworzenia aplikacji w Vue.js. Więcej na temat zasad użycia tych narzędzi znajdziesz w rozdziale 10. Na razie, aby uruchomić narzędzia deweloperskie, uruchom polecenie z listingu 9.5 w katalogu *nomagic*.

Listing 9.5. *Uruchamianie narzędzi deweloperskich*

```
npm run serve
```

W tym momencie nastąpi start serwera deweloperskiego HTTP. Otwórz nowe okno przeglądarki i przejdź pod adres *http://localhost:8080*. Zobaczysz treść zastępczą jak na rysunku 9.1.



Rysunek 9.1. *Uruchamianie przykładowej aplikacji*

Tworzenie aplikacji za pomocą API modelu DOM

Zaczniemy od utworzenia prostej aplikacji webowej bez użycia choćby odrobiny kodu Vue.js. Następnie pokażę, jak osiągnąć ten sam efekt za pomocą podstawowych opcji Vue.js.

W tym celu utworzę plik *main.js*, który zazwyczaj zawiera kod JavaScript odpowiedzialny za inicjalizację aplikacji Vue.js. Plik *main.js* to zwykły plik języka JavaScript, co oznacza, że możemy usunąć kod odpowiedzialny za konfigurację Vue.js i wstawić tam dowolne inne instrukcje.

W listingu 9.6 zamieniam domyślną treść pliku *main.js* na zwykłe instrukcje języka JavaScript, które korzystają z obiektowego modelu dokumentu (DOM). API modelu DOM oferują wszystkie przeglądarki w celu udostępniania treści dokumentu HTML w obrębie skryptów języka JavaScript. Model DOM stanowi podstawę działania praktycznie wszystkich aplikacji webowych.

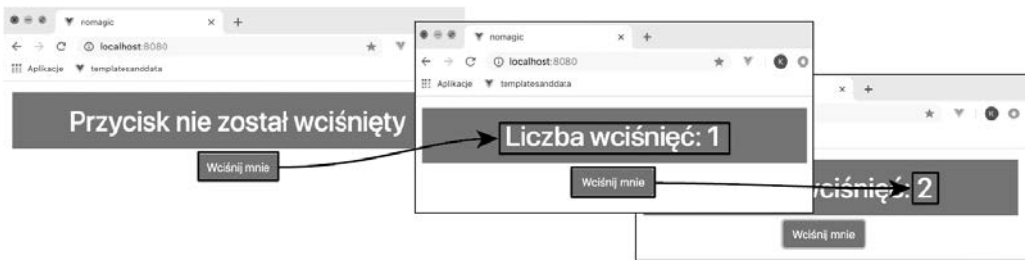
Listing 9.6. *Zamiana treści w pliku src/main.js*

```
require('../node_modules/bootstrap/dist/css/bootstrap.min.css')
let counter = 1;
let container = document.createElement("div");
container.classList.add("text-center", "p-3");
let msg = document.createElement("h1");
msg.classList.add("bg-primary", "text-white", "p-3");
```

```
msg.textContent = "Przycisk nie został wciśnięty";
let button = document.createElement("button");
button.textContent = "Wciśnij mnie";
button.classList.add("btn", "btn-secondary");
button.onclick = () => msg.textContent = `Liczba wciśnięć: ${counter++}`;
container.appendChild(msg);
container.appendChild(button);
let app = document.getElementById("app");
app.parentElement.replaceChild(container, app);
```

Nie martw się instrukcjami z powyższego kodu. W większości przypadków nie będziesz korzystać z API modelu DOM w projektach Vue.js. Celem tego przykładu jest przedstawienie pliku *main.js* jako zwykłego skryptu, zawierającego zwyczajny kod JavaScript, który może korzystać ze standardowych funkcji udostępnianych przez przeglądarkę.

Narzędzia deweloperskie uruchomione za pomocą polecenia z listingu 9.6 automatycznie wykrywają zmiany w plikach projektu. Zmiany są kompilowane, pakowane w jeden plik zawierający wszystkie funkcje aplikacji i wysyłane z powrotem do przeglądarki. Oznacza to, że po zapisaniu pliku *main.js* przeglądarka zostanie odświeżona, a Ty zobaczysz treść jak na rysunku 9.2. Na początku widać prosty komunikat, ale po kliknięciu przycisku zostaje on zastąpiony licznikiem. Licznik będzie zwiększać się po każdym wciśnięciu przycisku.



Rysunek 9.2. *Działanie bezpośrednio na modelu DOM*

Jak działa aplikacja w modelu DOM?

Z API modelu DOM korzystam w celu utworzenia elementu `div`, który zawiera elementy `h1` i `button`, w momencie wykonywania pliku *main.js*. Jednocześnie wiążę te elementy z klasami, które odpowiadają stylom zdefiniowanym we frameworku Bootstrap CSS, określającymi wygląd tych elementów. Następnie ustawiam słuchacza zdarzeń, który reaguje na kliknięcie przycisku poprzez modyfikowanie stanu licznika i wyświetlanie komunikatu w elemencie `h1`.

Po wykonaniu wyżej opisanych operacji lokalizuję istniejący w dokumencie element o ID równym `app` i zamieniam go na utworzony element `div`.

```
...
let app = document.getElementById("app");
app.parentElement.replaceChild(container, app);
...
```

Poniżej przedstawiam sposób, w jaki użytkownik otrzyma aplikację. Przeglądarka wysyła żądanie pod adres *http://localhost:8000*, a serwer deweloperski HTTP odpowiada zawartością pliku *index.html*:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width,initial-scale=1.0">
```

```
    <title>nomagic</title>
  </head>
  <body>
    <div id="app"></div>
  </body>
</html>
```

Zaznaczyłem wyraźnie element zamieniany przez kod w języku JavaScript. To właśnie tu znajdzie się kod HTML generowany w JavaScripcie.

Gdy serwer wygeneruje odpowiedź na żądanie HTTP przeglądarki, automatycznie zostanie wstawiony element `script`, który wczyta plik JavaScript zawierający cały kod projektu. Możesz to sprawdzić, klikając prawym przyciskiem myszy w oknie przeglądarki i wybierając z menu kontekstowego opcję *Wyświetl źródło strony*.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width,initial-scale=1.0">
    <title>nomagic</title>
  </head>
  <body>
    <div id="app"></div>
    <script type="text/javascript" src="/app.js"></script>
  </body>
</html>
```

Dostarczony plik JavaScript zawiera kod z pliku *main.js* i wszelkie jego zależności. Przeglądarka wykonuje instrukcje w pliku JavaScript, co prowadzi do wygenerowania omówionej wcześniej treści.

Kod HTML powstały w wyniku połączenia statycznych fragmentów z pliku *index.html* i dynamicznych znaczników wygenerowanych przez plik *main.js* można podejrzeć poprzez kliknięcie prawym przyciskiem w oknie przeglądarki i wybranie opcji *Zbadaj w menu kontekstowym*. W ten sposób zobaczymy widok na żywo dokumentu, utworzony za pomocą API modelu DOM.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width,initial-scale=1.0">
    <title>nomagic</title>
  </head>
  <body>
    <div class="text-center p-3">
      <h1 class="bg-primary text-white p-3">Przycisk nie został wciśnięty</h1>
      <button class="btn btn-secondary">Wciśnij mnie</button>
     </div>
    <script type="text/javascript" src="/app.js"></script>
  </body>
</html>
```

Efekt nie jest zbyt imponujący, ale w ten sposób pokazujemy, że zwykły kod JavaScript, z pomocą API modelu DOM, również może być używany do zamiany elementów w dokumentach HTML, a także do tworzenia nowej treści i reagowania na zdarzenia pochodzące od użytkownika. To są podstawy każdej aplikacji webowej — w tym także tej tworzonej w Vue.js.

Tworzenie obiektu Vue

Aplikacje Vue.js również korzystają z API modelu DOM do tworzenia treści HTML, reagowania na zdarzenia i aktualizowania wartości danych. Różnica polega na tym, że podejście z zastosowaniem Vue.js jest bardziej eleganckie, łatwiejsze do zrozumienia i bardziej skalowalne.

Przed chwilą skorzystałem z pliku *main.js* jako wygodnej metody wykonania kodu JavaScript, jednak jego głównym celem w aplikacji Vue.js jest utworzenie obiektu `Vue`, który stanowi punkt startowy całej aplikacji. W listingu 9.7 zamieniam kod API modelu DOM w pliku *main.js* na zestaw instrukcji, który tworzy obiekt `Vue`, i przywracam tym samym plikowi *main.js* jego pierwotne przeznaczenie.

Listing 9.7. *Tworzenie obiektu Vue w pliku src/main.js*

```
require('../node_modules/bootstrap/dist/css/bootstrap.min.css')
import Vue from "vue"
new Vue({
    el: "#app",
    template: `<div class="text-center p-3">
                    <h1 class="bg-secondary text-white p-3">
                        Vue: Przycisk nie został wciśnięty
                    </h1>
                    <button class="btn btn-secondary">
                        Wciśnij mnie
                    </button>
</div>`
});
```

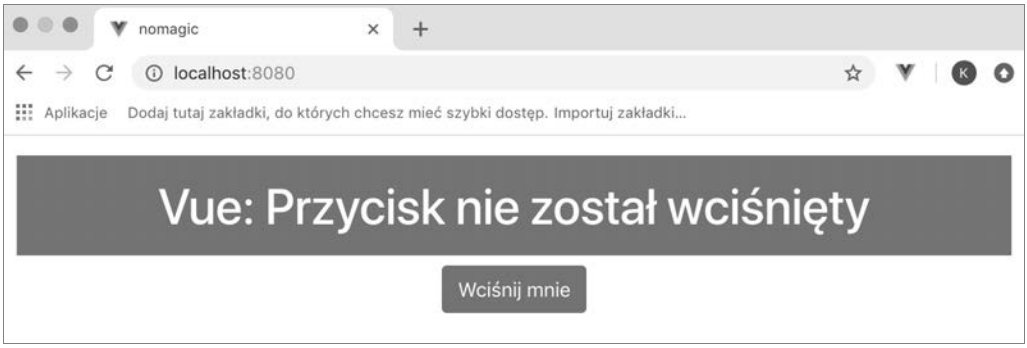
Instrukcja `import` jest niezbędna do zaimportowania obiektu `Vue` z modułu `vue`, pobranego do katalogu *node_modules* w momencie utworzenia projektu. Obiekt `Vue` został utworzony za pomocą słowa kluczowego `new`, a konstruktor przyjął obiekt konfiguracji, którego właściwości pozwalają na kontrolę zachowania aplikacji i definiują treści przedstawiane użytkownikowi.

W tym przykładzie mamy do czynienia z dwoma właściwościami konfiguracji: `el` i `template`. Obiekt `Vue` korzysta z właściwości `el`, aby zidentyfikować element, który zostanie zamieniony przez zawartość aplikacji w pliku *index.html*. Właściwość `template` dostarcza zawartość HTML, która zamieni dotychczasową zawartość elementu ukrytego za właściwością `el`. Jak pewnie pamiętasz, te zadania musiałem z pomocą API modelu DOM wykonywać ręcznie w poprzednim przykładzie.

W listingu 9.7 ustawiam właściwość `el` na wartość `#app`, która pobiera element o `id` równym `app`. Ustawiam właściwość `template` w taki sposób, aby zawierała tę samą strukturę kodu HTML co w listingu 9.6. Tym razem mogę jednak po prostu napisać kod HTML, zamiast tworzyć identyczną konstrukcję za pomocą wywołań kodu JavaScript. Obiekt konfiguracji jest zdefiniowany w kodzie JavaScript, co oznacza, że kod HTML w nim osadzony musi być zawarty jako prawidłowy łańcuch znaków JavaScript. Skorzystałem z grawisa (znaku `), dzięki czemu jestem w stanie pisać kod w wielu wierszach, przez co jest on łatwiejszy do odczytu.

■ **Ostrzeżenie** Możesz korzystać z właściwości `template` tylko, jeśli przesłoniłeś ustawienia projektu jak w listingu 9.2. Domyślnie funkcja do przetwarzania tekstów zawierających szablony jest wyłączona w Vue.js.

Po zapisaniu zmian w pliku *main.js* przeglądarka zostanie odświeżona, a treść będzie przypominać tę z rysunku 9.3. Zmieniła się treść komunikatu, a także kolor tła.



Rysunek 9.3. *Zastosowanie obiektu Vue*

Stosowanie obiektu Vue

Mój obiekt `Vue` wyświetla zawartość HTML użytkownikowi, ale to tylko część funkcji, które są mi potrzebne. Kolejnym krokiem jest dodanie danych i wyświetlenie ich użytkownikowi. W listingu 9.8 dodaję zmienną `counter` i modyfikuję łańcuch `template` tak, aby został wyświetlony użytkownikowi.

Listing 9.8. *Dodawanie zmiennej do pliku src/main.js*

```
require('../node_modules/bootstrap/dist/css/bootstrap.min.css')
import Vue from "vue"
new Vue({
    el: "#app",
    template: `<div class="text-center p-3">
                    <h1 class="bg-secondary text-white p-3">
                        Liczba wciśnięć: {{ counter }}
                    </h1>
                    <button class="btn btn-secondary">
                         Wciśnij mnie
                    </button>
                </div>`,
    data: {
        counter: 0
    }
});
```

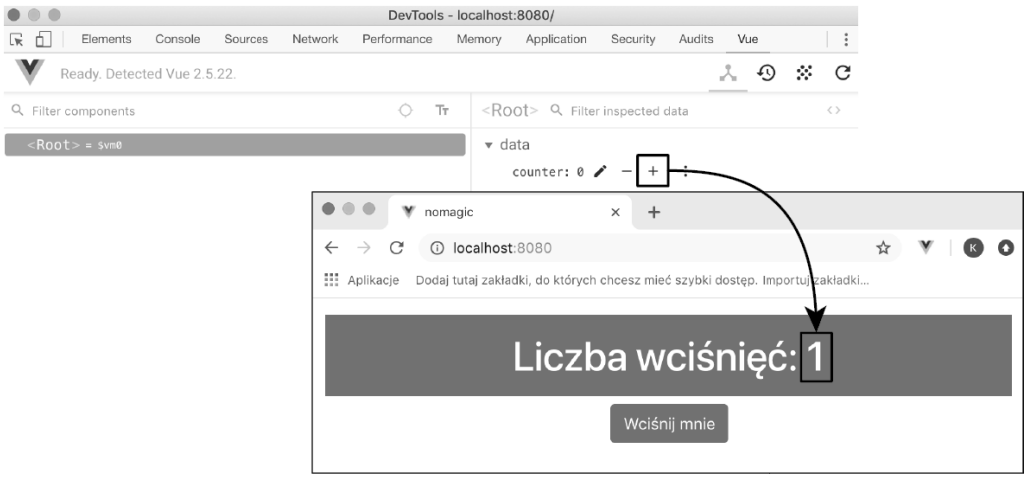
Dodałem właściwość `data` do obiektu konfiguracyjnego aplikacji. Obiekt definiuje właściwość `counter` z wartością początkową równą zero. Aby wyświetlić wartość licznika, stosuję proste wiązanie danych w elemencie `h1`.

```
...
<h1 class="bg-secondary text-white p-3">
    Liczba wciśnięć: {{ counter }}
</h1>
...
```

Wiązania danych to niezwykle ważna funkcja Vue.js, która łączy obiekt `data` z treścią właściwości `template`. Gdy Vue.js wyświetla treść szablonu, poszukuje w nim wiązań danych, rozpoznaje je jako wyrażenia języka JavaScript i dołącza wynik operacji jako kod HTML. W tym przypadku wiązanie danych odwołuje się do jednej z właściwości obiektu `data`. W rezultacie wartość zmiennej `counter` zostanie wstawiona na stronę w elemencie `h1`.

Dane są **reaktywne**, **odświeżane na żywo** — oznacza to, że zmiana wartości właściwości `counter` zostanie automatycznie odzwierciedlona w elemencie `h1`. Po utworzeniu obiektu `Vue` następuje przetworzenie obiektu `data` i zamiana właściwości, przez co możliwe jest wykrycie zmian w ich wartościach.

Konsekwencję reaktywności danych widać w narzędziu Vue Devtools. Otwórz narzędzia deweloperskie za pomocą klawisza *F12*, przejdź na zakładkę *Vue* i kliknij element `<Root>`. W panelu po prawej stronie przesuń kursor na element `counter` i kliknij przycisk +, aby zwiększyć jego wartość. Vue.js wykryje zmianę wartości właściwości i przetworzy kod szablonu ponownie, generując efekt jak na rysunku 9.4. Szczegóły instalacji tej wtyczki znajdziesz na stronie *https://github.com/vuejs/vue-devtools*.



Rysunek 9.4. *Zmiana wartości zmiennej reaktywnej*

Dodawanie funkcji obsługi zdarzenia

Kolejnym krokiem w tworzeniu tego przykładu jest zwiększenie wartości zmiennej `counter` automatycznie po kliknięciu przycisku. Poprzednio mogłem powiązać funkcję obsługi zdarzenia bezpośrednio za pomocą instrukcji języka JavaScript. Tym razem zastosuję inne podejście (listing 9.9).

Listing 9.9. *Obsługa zdarzenia w pliku src/main.js*

```
require('../node_modules/bootstrap/dist/css/bootstrap.min.css')
import Vue from "vue"
new Vue({
    el: "#app",
    template: `<div class="text-center p-3">
                    <h1 class="bg-secondary text-white p-3">
                        Liczba wciśnięć: {{ counter }}
                    </h1>
                    <button class="btn btn-secondary" v-on:click="handleClick">
                         Wciśnij mnie
                    </button>
                </div>`,
    data: {
        counter: 0
    },
    methods: {
        handleClick() {
            this.counter++;
```

```
        }
      }
});
```

Atrybut, który dodaję do elementu `button`, to **dyrektywa**. Jest to mechanizm Vue.js, który pozwala na rozszerzenie funkcjonalności elementu HTML. W tym przypadku skorzystałem z dyrektywy `v-on`, aby umożliwić obsługę zdarzenia `click` przycisku. Wiążę je z metodą `handleClick`, która została zadeklarowana we właściwości `methods` obiektu konfiguracyjnego `Vue`. Obiekt `methods` definiuje funkcję `handleClick`, która zwiększa wartość zmiennej `counter`.

Skoro Vue.js działa reaktywnie, zmiana wartości zmiennej `counter` pociąga za sobą ponowną ewaluację wiązania danych w elemencie `h1`, a w konsekwencji — zmianę komunikatu wyświetlanego użytkownikowi (rysunek 9.5).

Rysunek 9.5. *Obsługa zdarzenia*

Modyfikacja komunikatu

Ostatnią zmianą niezbędną do zrównania tego przykładu z poprzednim jest wyświetlenie innego komunikatu, gdy użytkownik nie zdąży jeszcze kliknąć przycisku. Aby osiągnąć ten sam efekt za pomocą obiektu `Vue`, wprowadziłem zmiany w listingu 9.10.

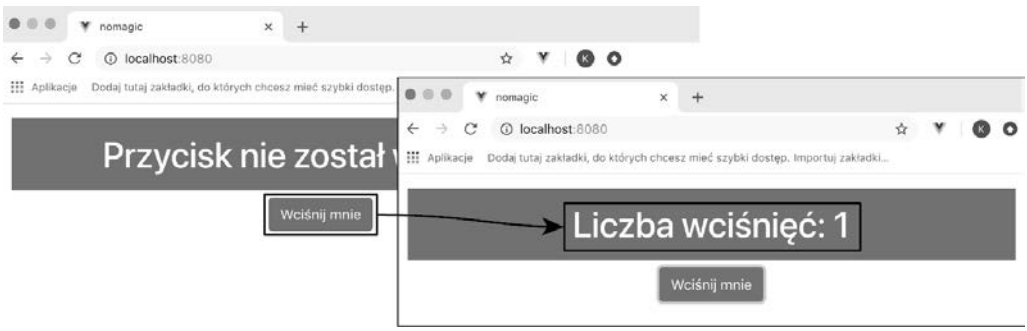
Listing 9.10. *Wyświetlanie warunkowego komunikatu w pliku src/main.js*

```
require('../node_modules/bootstrap/dist/css/bootstrap.min.css')
import Vue from "vue"
new Vue({
    el: "#app",
    template: `<div class="text-center p-3">
                    <h1 class="bg-secondary text-white p-3">
                        {{ message }}
                    </h1>
                    <button class="btn btn-secondary" v-on:click="handleClick">
                         Wciśnij mnie
                    </button>
                </div>`,
```

```
    data: {
         counter: 0
    },
    methods: {
        handleClick() {
            this.counter++;
        }
    },
    computed: {
        message() {
            return this.counter == 0 ?
                "Przycisk nie został wciśnięty" : `Liczba wciśnięć: ${this.counter}`;
          }
      }
});
```

Właściwość `computed` określa wartości danych, które wymagają dokonania pewnych operacji (często obliczeń), aby uzyskać rezultat. W tym przykładzie korzystamy z wyrażenia, aby sprawdzić wartość właściwości `counter` i na podstawie tego sprawdzenia zwracamy tekst. Nowo obliczona właściwość nazywa się `message` i wyświetlamy jej wartość w elemencie `h1`, umieszczając nazwę w wiązaniu danych, tak jak wcześniej zrobiliśmy to ze zmienną `counter`.

Gdy użytkownik klika przycisk, dyrektywa `v-on` wywołuje metodę `handleClick`, która zwiększa wartość zmiennej `counter`. Vue.js wykrywa zmianę i ponownie ewaluuje wyrażenia wiązań danych zdefiniowane w treści szablonu. Otrzymujemy nową wartość dla obliczonej właściwości, która to wartość jest wyświetlana użytkownikowi za pomocą elementu `h1` (rysunek 9.6).

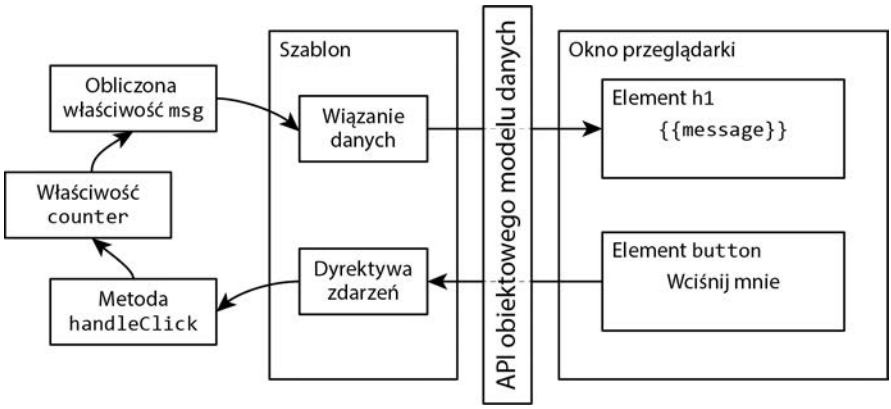


Rysunek 9.6. *Zastosowanie właściwości obliczanej*

Zasada działania obiektu Vue

Obiekt `Vue` ma taki sam zakres funkcjonalny jak kod zastosowany w listingu 9.6, jednak dzięki Vue.js aplikacja ma znacznie lepszą strukturę, co pozwala na łatwiejsze zarządzanie i rozwijanie oprogramowania niż w przypadku pracy z modelem DOM. Trudno docenić Vue.js, patrząc tylko na obiekt z listingu 9.10, jednak już w tym przykładzie widać strukturę, w której istnieje podział na treść, logikę i dane aplikacji jak na rysunku 9.7.

Szablon obiektu `Vue` dostarcza treść HTML, którą widzi użytkownik. Ponadto otrzymujemy wiązanie danych i dyrektywę zdarzenia, która zapewnia interaktywność aplikacji. Dane aplikacji zawierają właściwość `counter` i właściwość obliczoną `msg`, a logika, która łączy je ze sobą, jest zawarta w metodzie `handleClick`.



Rysunek 9.7. *Struktura aplikacji*

Komponenty w praktyce

Obiekt `Vue` pomaga oddzielić różne aspekty aplikacji, jednak rezultat wcale nie jest satysfakcjonujący. W większości projektów obiekt `Vue` jest używany do skonfigurowania aplikacji, ale jej zasadnicze funkcje są definiowane za pomocą **komponentów**, które rozszerzają możliwości obiektu `Vue`. Ponadto komponenty mogą być definiowane w plikach zawierających użyteczną mieszankę różnego rodzaju treści.

Komponenty definiuje się w plikach z rozszerzeniem *.vue*. Tuż po utworzeniu projekt zawiera plik *App.vue*; w listingu 9.11 przedstawiam jego nową zawartość.

Listing 9.11. *Zamiana zawartości pliku src/App.vue*

```
<script>
export default {
    template: `<div class="text-center p-3">
                    <h1 class="bg-secondary text-white p-3">
                        {{ message }}
                    </h1>
                    <button class="btn btn-secondary" v-on:click="handleClick">
                         Wciśnij mnie
                    </button>
                </div>`,
    data: function () {
        return {
            counter: 0
        }
   },
   methods: {
        handleClick() {
            this.counter++;
        }
    },
    computed: {
        message() {
            return this.counter == 0 ?
                "Przycisk nie został wciśnięty" : `Liczba wciśnięć: ${this.counter}`;
         }
     }
  }
</script>
```

Komponent przedstawiony w listingu 9.11 zawiera te same funkcje co w przypadku definicji w obiekcie `Vue`, z wyjątkiem tego, że właściwości języka JavaScript są zdefiniowane w elemencie `script`. Na pierwszy rzut oka może wydawać się, że nie jest to duże usprawnienie, ale jest to dobry przykład na to, że komponenty są obiektami `Vue`, podobnymi do tego z pliku *main.js*. Komponenty pozwalają na tworzenie aplikacji z wielu komponentów, które można łączyć, aby zapewnić użytkownikowi zaawansowane możliwości.

Kod w języku JavaScript jest zawarty w elemencie `script` z tym samym zakresem funkcjonalnym co obiekt `Vue`. W porównaniu z wersją ze zwykłym obiektem `Vue` są dwie zmiany. Pierwsza z nich dotyczy konieczności wyeksportowania obiektu konfiguracji:

```
...
export default {
    // …właściwości konfiguracji zostały pominięte…
}
...
```

W ten sposób obiekt może zostać zaimportowany ze swojego modułu, co osiągamy w listingu 9.12. Druga zmiana polega na wyrażeniu właściwości `data` w formie funkcji:

```
...
data: function () {
    return {
       counter: 0
    }
},
...
```

Właściwość `data` otrzymuje funkcję, która zwraca obiekt. Właściwości tego obiektu dostarczają wartości danych dla komponentu. Jest to nieco dziwne podejście, ale Vue.js wymaga tego, aby zapewnić, że każdy komponent dysponuje własnymi danymi.

Rejestracja i wdrażanie komponentu

Komponenty należy zarejestrować w Vue.js przed pierwszym użyciem. Z reguły odbywa się to w pliku *main.js*. W listingu 9.12 zamieniam konfigurację obiektu `Vue` w taki sposób, aby skorzystać z nowego komponentu.

Listing 9.12. *Konfiguracja komponentu w pliku src/main.js*

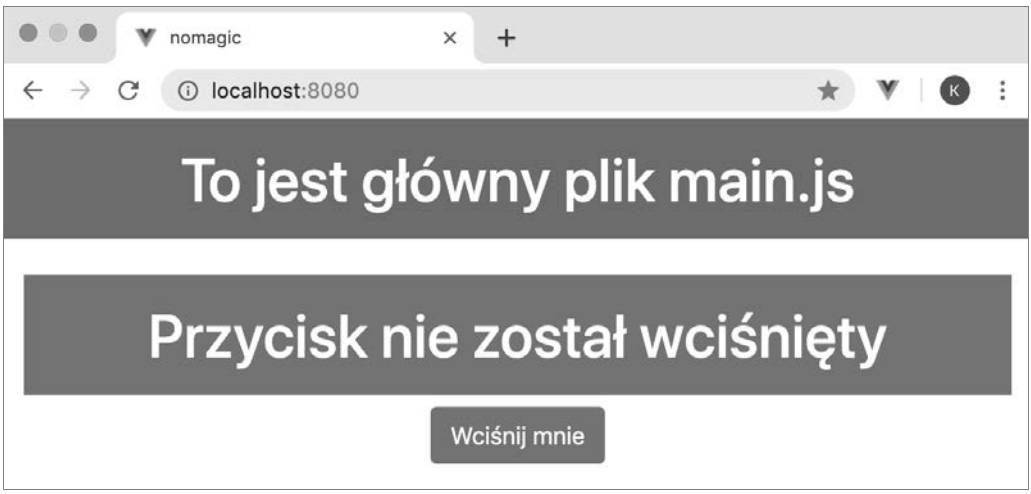
```
require('../node_modules/bootstrap/dist/css/bootstrap.min.css')
import Vue from "vue";
import MyComponent from "./App";
new Vue({
    el: "#app",
    components: { "custom": MyComponent },
    template: `<div class="text-center">
                        <h1 class="bg-primary text-white p-3">
                          To jest główny plik main.js
                        </h1>
                        <custom />
                </div>`
});
```

Pierwsza nowa instrukcja korzysta ze słowa kluczowego `import`, które importuje moduł z pliku *App.vue* i nadaje mu nazwę `MyComponent`. Zgodnie z konwencją nazwa pliku powinna być używana jako nazwa importu, ale możesz skorzystać z takiej nazwy, która wydaje Ci się najbardziej sensowna.

Kolejna zmiana polega na dodaniu właściwości `components` do konfiguracji obiektu `Vue`. Ta właściwość powoduje otrzymanie obiektu, który dostarcza do Vue.js listę elementów HTML i skojarzonych z nimi komponentów. W tym przypadku skonfigurowałem właściwość `components` tak, aby element `custom` został skojarzony z komponentem `MyComponent`, do którego odwołuję się w pliku *App.vue*.

Zmieniłem także treść szablonu (`template`), dzięki czemu zawiera on prostszy zbiór elementów HTML, wśród których zawarty jest element `custom`.

Gdy obiekt `Vue` podczas przetwarzania swojego szablonu napotka element `custom`, element ten zostanie zastąpiony przez treść zdefiniowaną w ramach komponentu w pliku *App.vue*. W rezultacie użytkownik otrzyma połączenie kodu HTML z właściwości `template` komponentu, a także z właściwości `template` obiektu `Vue`. Po zapisaniu zmian przeglądarka zostanie odświeżona, a Ty zobaczysz treść jak na rysunku 9.8.



Rysunek 9.8. *Komponent w praktyce*

Obiekt `Vue` zazwyczaj nie wyświetla swojej własnej treści. Zgodnie z konwencją kod HTML umieszcza się w szablonie komponentu, jak w listingu 9.13.

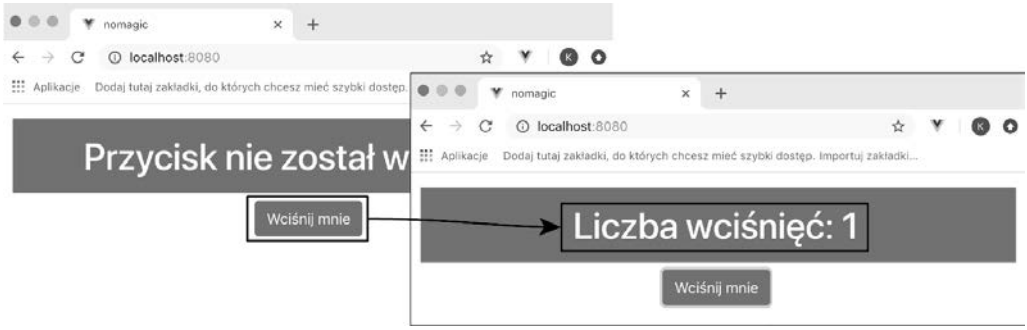
Listing 9.13. *Usuwanie treści szablonu w pliku src/main.js*

```
require('../node_modules/bootstrap/dist/css/bootstrap.min.css')
import Vue from "vue";
import MyComponent from "./App";
new Vue({
    el: "#app",
    components: { "custom": MyComponent },
    template: "<custom />"
});
```

Obiekt `Vue` zastępuje element `div` w pliku *index.html* treścią elementu `custom`, zadeklarowaną w elemencie `template` komponentu. Po zapisaniu zmian zobaczysz kompletny przykład aplikacji, jak na rysunku 9.9.

Oddzielanie szablonu od kodu JavaScript

Szablony komponentu można definiować za pomocą właściwości `template`, jednak warto rozważyć inne podejście, które preferuje wielu programistów. W listingu 9.14 usuwam właściwość `template` i zastępuję ją elementem `template`, zgodnie z typową metodą deklarowania komponentów.



Rysunek 9.9. *Kompletna przykładowa aplikacja*

Listing 9.14. *Zastosowanie elementu template w pliku src/App.vue*

```
<template>
    <div class="text-center p-3">
      <h1 class="bg-secondary text-white p-3">
        {{ message }}
      </h1>
      <button class="btn btn-secondary" v-on:click="handleClick">
          Wciśnij mnie
      </button>
    </div>
</template>
<script>
export default {
    data: function () {
        return {
          counter: 0 }
        },
    methods: {
        handleClick() {
            this.counter++;
        }
    },
    computed: {
        message() {
            return this.counter == 0 ?
                "Przycisk nie został wciśnięty" : `Liczba wciśnięć: ${this.counter}`;
        }
     }
 }
</script>
```

Element `template` zawiera tę samą treść co właściwość `template`, ale to podejście pozwala na traktowanie kodu w większości edytorów jako HTML, co owocuje wsparciem na poziomie edytora, takim jak autouzupełnianie nazw czy oznaczanie błędów. Nic się nie zmienia w wyglądzie i zachowaniu — ta zmiana ma na celu uczynienie procesu deweloperskiego łatwiejszym.

Tworzenie odrębnych plików JavaScript i HTML

Nie każdy lubi łączyć kod JavaScript z HTML-em w tym samym pliku. Jeśli tak jest i w Twoim przypadku, to możesz tworzyć odrębne pliki na oba rodzaje kodu. W listingu 9.15 przedstawiam przykład pliku zawierającego wyłącznie kod HTML.

Listing 9.15. *Zawartość pliku src/App.html*

```
<div class="text-center p-3">
    <h1 class="bg-secondary text-white p-3">
      {{ message }}
    </h1>
    <button class="btn btn-secondary" v-on:click="handleClick">
        Wciśnij mnie
    </button>
</div>
```

W listingu 9.16 usuwam zawartość elementu `template` i dodaję atrybut `src`, który informuje Vue.js, że treść komponentu można znaleźć w odrębnym pliku HTML.

Listing 9.16. *Odwołanie do pliku HTML w pliku src/App.vue*

```
<template src="./App.html" />
<script>
export default {
    data: function () {
        return {
counter: 0 }
}, methods: {
        handleClick() {
            this.counter++;
          }
       },
    computed: {
        message() {
            return this.counter == 0 ?
                "Przycisk nie został wciśnięty" : `Liczba wciśnięć: ${this.counter}`;
        }
    }
}
</script>
```

Podsumowanie

W tym rozdziale wyjaśniłem, że Vue.js korzysta z tego samego API modelu DOM do tworzenia aplikacji webowych, które może być używane w czystym, „waniliowym” JavaScripcie. Za Vue.js nie kryje się bowiem żadna magia. Jak przekonasz się w kolejnych rozdziałach, choć można bezpośrednio korzystać z API modelu DOM, Vue.js zapewnia dużo przydatnych funkcji, które pozwalają na tworzenie bardziej eleganckich, możliwych do zarządzania i skalowalnych funkcji. W kolejnym rozdziale objaśnię strukturę projektów Vue.js i działanie narzędzi deweloperskich.

Skorowidz

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

V

W

X

Z

Ż

Vue.js 2: zyskaj większe możliwości i pisz najlepsze aplikacje!

Vue.js jest frameworkiem, który służy do budowy nowoczesnych, reaktywnych i skalowanych aplikacji WWW, a przy tym ułatwia korzystanie z komponentów wielokrotnego użytku. Framework ten powstał jako narzędzie do szybkiego prototypowania, a teraz dynamicznie się rozwija i ewoluuje, dzięki czemu liczba dostępnych funkcji stale rośnie. Prostocie i wszechstronności zawdzięcza ogromne uznanie deweloperów. Z całą pewnością można go uznać za narzędzie niezbędne każdemu, kto buduje kompleksowe aplikacje WWW i pragnie utrzymywać wysokie standardy.

Dzięki tej książce dowiesz się, czym jest Vue.js i jak rozpocząć z nim pracę, przekonasz się także, jakie ma możliwości. Nauczysz się budować dynamiczne aplikacje wykorzystujące właściwości nowoczesnych przeglądarek internetowych i urządzeń. Odkryjesz zalety wzorca MVC (model – widok – kontroler) i dowiesz się, jak zadziwiająco sprawnym językiem stał się JavaScript. Zapoznasz się z anatomią projektu Vue.js, z procesami kompilacji i transformacji oraz nabierzesz biegłości w posługiwaniu się zaawansowanymi funkcjami frameworka. Liczne przykłady pozwolą Ci na niemal natychmiastowe przetestowanie opisywanych zagadnień w praktyce. W rezultacie bardzo szybko będziesz przygotowany do tworzenia zaawansowanych, reaktywnych i dynamicznych aplikacji WWW!

W tej książce między innymi:

- zarys budowy frameworka, jego instalacja i sposoby wykorzystania
- wzorzec MVC i architektura aplikacji w Vue.js 2
- dynamiczne aplikacje WWW po stronie klienta
- korzystanie z usług REST
- rozszerzanie i modyfikowanie Vue.js w zależności od potrzeb

Adam Freeman — jest doświadczonym specjalistą IT. Kierował zespołami inżynierów w wielu firmach, ostatnio jako dyrektor ds. technologicznych i dyrektor ds. operacyjnych w międzynarodowym banku. Po przejściu na emeryturę poświęca czas na pisanie znakomitych książek i biegi długodystansowe.

ISBN 978-83-283-5498-2